Advanced Programming Topics

Why functional programming? Why algorithms? Why complexity? Why proof?
Lecture 1

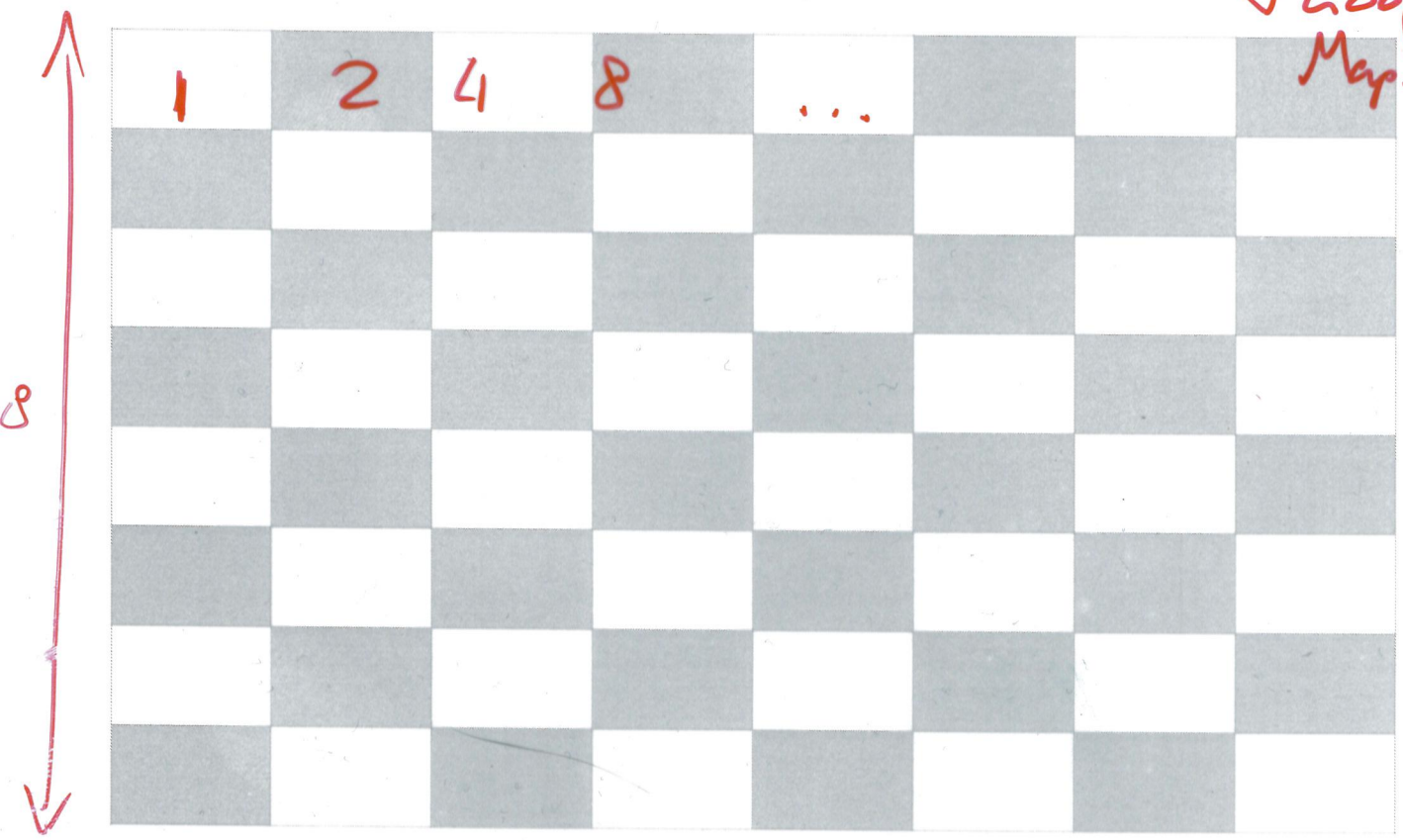
James Marshall

Why functional programming?

focus on the algorithm

real world impact ⇒ Java 8
⇒ Mathematica
⇒ Google MapReduce

The inventor and the king



$$\sum_{i=0}^{63} 2^i = 18.4 \times 10^{18}$$

closed-form

one grain of rice = 2×10^{-2} kg

$\approx 37 \times 10^{16}$ kg ↑ 1m years

annual Chinese rice production 20.2×10^{10} kg

A famous number sequence

1 1 2 3 5 8 13 21 34 ...

Fibonacci

A Haskell implementation

```
f 1 = 1
f 2 = 1
f n = f (n-1) + f (n-2)
```

1st number in seq. is 1
2nd " " " " "

Q: How long would it take to compute `f 300` *?*

$\simeq 1 \times 10^{16}$ years assuming 1m operations per second

Q: How old is the universe estimated to be?

$\simeq 13.8 \times 10^{9}$ years

A better Haskell implementation

```
g 1 a b = a
g n a b = g (n-1) b (a+b)

h n = g n 1 1
```

} memorization

Q: How long would it take to compute `h 1000` *?*

Not long

Q: How do we work out the answers to these kind of questions?

[asymptotic complexity analysis

Q: What makes `h` *faster than* `f` *?*

algorithm design (dynamic programming)

Q: How do we know our algorithms are working correctly?

proof

Advanced Programming Topics

Asymptotic complexity analysis
Lecture 2
James Marshall

input size

Asymptotic notation (from COM2003)

$O(g(n))$	— 'big-oh' —	upper bound
$\Omega(g(n))$	— 'big-omega' —	lower bound
$\Theta(g(n))$	— 'big-theta' —	upper and lower bound

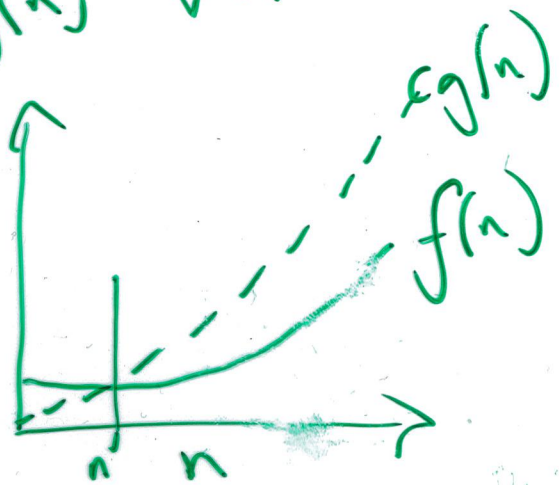
for all $n > n'$

member of

Upper bounds

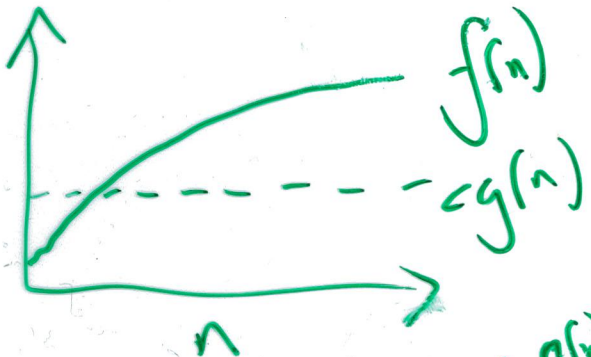
$f(n) \in O(g(n)) \Rightarrow$ $0 \leq f(n) \leq cg(n) \quad \forall n > n'$

run time — bound



Lower bounds

$f(n) \in \Omega(g(n)) \Rightarrow$ $0 \leq cg(n) \leq f(n)$

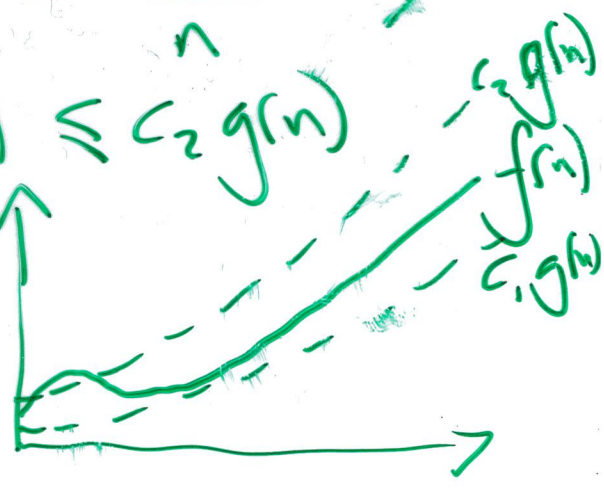


Upper and lower bounds

$f(n) \in \Theta(g(n)) \Rightarrow$ $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

$f(n) = 100 - 4n + n^2$

$f(n) = 100 - 2n^2 + n^3$



Worst-case vs average-case complexity

Usually it is far easier to work out the maximum number of steps an algorithm will take, rather than the average number, since we then need to know something about how probable different inputs to the algorithm are. Asymptotic complexities normally refer to worst-case complexity.

Best and worst cases for some sorting algorithms

```
--insertion sort (from Thompson, The Craft of Functional
Programming)
iSort :: Ord a => [a] -> [a]
iSort []     = []
iSort (x:xs) = ins x (iSort xs)

ins :: Ord a => a -> [a] -> [a]
ins x []     = [x]
ins x (y:ys)
  | x <= y    = x:(y:ys)
  | otherwise = y : ins x ys
```

empty list is sorted

$iSort \in \Omega(n)$ (best case)

$iSort \in O(n^2)$ (worst case)

```
--mergesort (from Thompson, The Craft of Functional Programming,
with merge function added by J. A. R. Marshall)
mergeSort :: Ord a => [a] -> [a]
mergeSort xs
  | length xs < 2 = xs
  | otherwise
     = merge (mergeSort first) (mergeSort second)
       where
       first = take half xs
       second = drop half xs
       half = (length xs) `div` 2

merge :: Ord a => [a] -> [a] -> [a]
merge [] [] = []
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys)
  | x < y  = [x]++(merge xs (y:ys))
  | y < x  = [y]++(merge (x:xs) ys)
  | x == y = [x]++[y]++(merge xs ys)
```

$mergesort \in \Theta(n \log n)$

empty list is sorted

base of log irrelevant

```
--quicksort (naive) (from Thompson, The Craft of Functional
Programming)
qSort :: Ord a => [a] -> [a]
qSort [] = []
qSort (x:xs) = qSort [y | y<-xs, y<=x] ++ [x] ++ qSort [y | y<-xs,
     y>x]
```

$qSort \in O(n^2)$ (worst case)

Advanced Programming Topics

Algorithm analysis and design - divide-and-conquer
Lecture 3
James Marshall

Divide-and-conquer algorithms

Divide-and-conquer algorithms work by taking their input, dividing it into smaller sub-problems, conquering these independently, then combining the results into a single solution for output. Most functional programs are divide-and-conquer, since recursion works by repeatedly breaking problems down into simpler problems, until the simplest possible problem is found for which a solution is known, then combining the resulting solutions.

$O(n)$, $\Omega(n)$, $\Theta(n)$

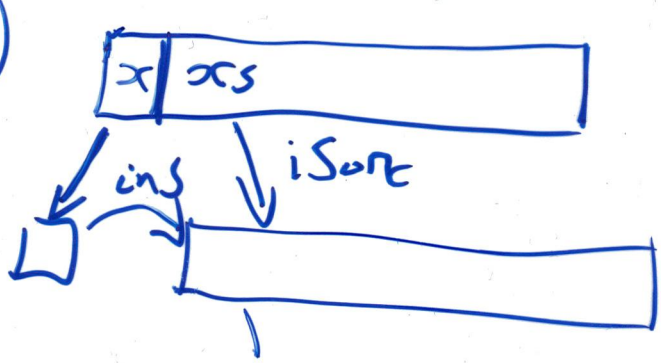
Analysis - insertion sort

```
--insertion sort (from Thompson, The Craft of Functional
Programming)
iSort :: Ord a => [a] -> [a]
iSort []     = []
iSort (x:xs) = ins x (iSort xs)

ins :: Ord a => a -> [a] -> [a]
ins x []     = [x]
ins x (y:ys)
 | x <= y    = x:(y:ys)
 | otherwise = y : ins x ys
```

$\in \Omega(n)$ (sorted list)

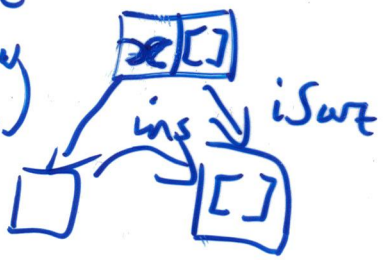
and $\in O(n^2)$
(reverse-sorted list)

constant constant
n(iSort + ins)



Each iSort takes constant time

Each ins takes (for list of length i)
constant time
(new element is new head)
i steps



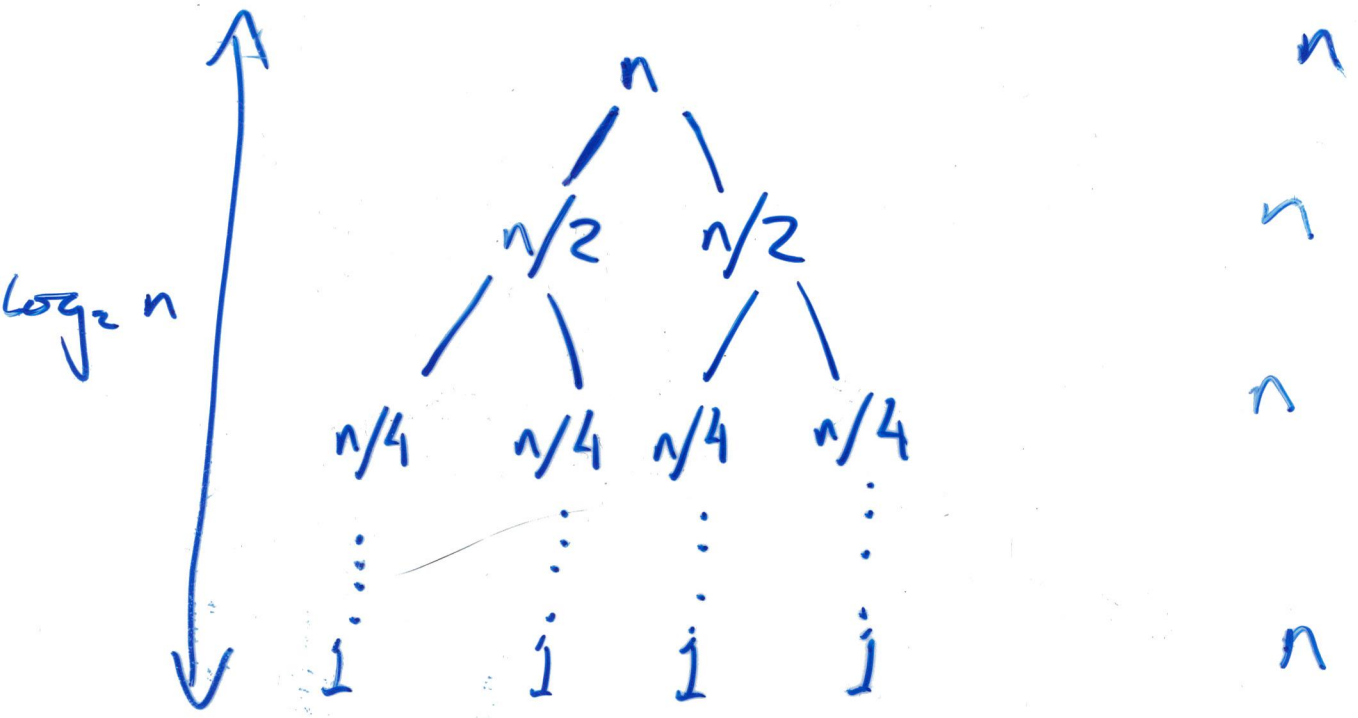
Analysis - mergesort

```
--mergesort (from Thompson, The Craft of Functional Programming,
with merge function added by J. A. R. Marshall)
mergeSort :: Ord a => [a] -> [a]
mergeSort xs
  | length xs < 2 = xs
  | otherwise
     = merge (mergeSort first) (mergeSort second)
       where
       first = take half xs
       second = drop half xs
       half = (length xs) `div` 2

merge :: Ord a => [a] -> [a] -> [a]
merge [] [] = []
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys)
  | x < y  = [x]++(merge xs (y:ys))
  | y < x  = [y]++(merge (x:xs) ys)
  | x == y = [x]++[y]++(merge xs ys)
```

$\in \Theta(n \log n)$

why?

Recurrence tree:



Splitting/recombination cost = c

$\Rightarrow$ total cost $c \times \log_2 n \times n$

$\in \Theta(n \log n)$

Analysis - quicksort

```
--quicksort (naive) (from Thompson, The Craft of Functional
Programming)
qSort :: Ord a => [a] -> [a]
qSort [] = []
qSort (x:xs) = qSort [y | y<-xs, y<=x] ++ [x] ++ qSort [y | y<-xs,
     y>x]
```

Best case : $\Omega(n \log n)$

(like mergesort)

Worst case: $O(n^2)$

(like insertion sort)

Bad pivot:

$x \mid xs \geq x$

$x \mid xs \leq x$

} Unlikely

Good pivot:

$x \mid xs$ (some $< x$, some $\geq x$) More likely

Average-case complexity of qSort

Despite `qSort` having a worst-case complexity of order $O(n^2)$ it is one of the most popular sorting algorithms used for large lists. Why?

Intuition: The most efficient way for `qSort` to proceed is to split the list it is sorting into two equally sized sub-lists for sorting, with the first sublist containing only elements smaller than the 'pivot', and the second sublist containing only elements that are larger.

Consider what happens in the 'average case' of a list of uniformly-distributed random numbers. At every divide step the pivot chosen by the naive version of `qSort` above is the head of the list:

Because of this in the average case `qSort` behaves like `mergeSort`, and so has average-case asymptotic complexity in $O(n \log n)$ (assuming that uniformly-randomly distributed lists represent the average case)

Other reasons for quicksort's popularity

- randomise selection of pivot
- in-place sorting (in practice)

Advanced Programming Topics

Solving recurrences and the Master Theorem
Lecture 4

James Marshall

open-form

Q: In lecture 3, how did we know that

$$\sum_{i=1}^{n} (k_1(i-1)+k_2) = \frac{k_1 n(n+1)}{2} + n(k_2-k_1)$$

Q: is there a general method for analysing divide-and-conquer algorithms?

Sort of...

Some tricks for solving recurrences

In counting the steps involved in executing an algorithm, generally we end up with open-form solutions involving summations, *e.g.*

$$\sum_{i=1}^{n} i = 1+2+3+\ldots+n =$$

With some know-how, these can typically be reduced to closed-form solutions suitable for asymptotic analysis. For instance, the example above can be shown to be in

$$\Theta(n^2)$$

But how? Gauss worked this one out when he was a schoolboy...

$$1+2+3+\ldots(n-2)+(n-1)+n = (n+1)\frac{n}{2}$$

Solving arithmetic series (in general)

$$\sum_{x=1}^{n} (a+bx) = na + \frac{n(n+1)}{2} b \in \Theta(n^2)$$

i.e. polynomial

Solving geometric series (e.g. lecture 1)

$$\sum_{x=0}^{n} a^x = 1+a+a^2 \ldots a^n = \frac{a^{n+1}-1}{a-1} \in \Theta(a^n)$$

Other tricks available, see e.g. CLRS appendix A

The Master Theorem (e.g. Theorem 4.1 in CLRS)

For constants $a \geq 1$, $b > 1$ and for $n \geq 0$, let the running time $T(n)$ of an algorithm be defined by the recurrence

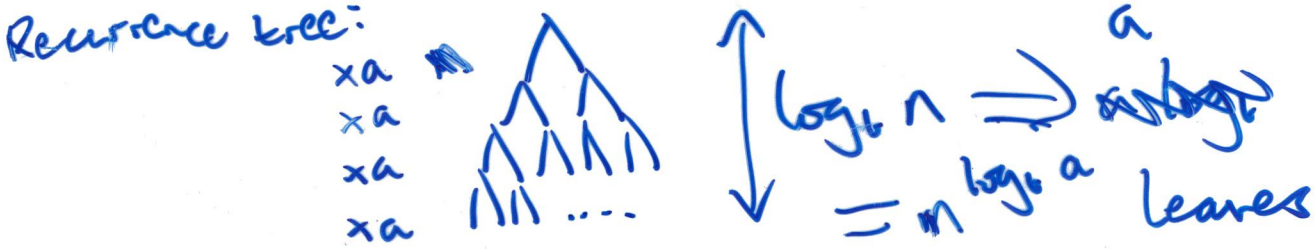
$$T(n) = a\,T(n/b) + f(n)$$

Then we can work out an asymptotic bound for $T(n)$ if one of the following three conditions holds:

1. If $f(n) \in O\left(n^{\log_b a - \varepsilon}\right)$ for some constant $\varepsilon > 0$, then $T(n) \in \Theta\left(n^{\log_b a}\right)$
2. If $f(n) \in \Theta\left(n^{\log_b a}\right)$ then $T(n) \in \Theta\left(n^{\log_b a} \log n\right)$
3. If $f(n) \in \Omega\left(n^{\log_b a + \varepsilon}\right)$ for some constant $\varepsilon > 0$, and if $a\,f(n/b) \leq c\,f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) \in \Theta(f(n))$

Explaining the Master Theorem

What is $n^{\log_b a}$? It is the number of leaves in the recurrence tree. Why?



So the Master Theorem compares the work done in 'bookkeeping' (splitting into subproblems, and then combining solutions) against the amount of work done solving the simplest subproblems, the leaves:

1. 'Cheap' bookkeeping - dividing and combining is *polynomially cheaper* than the work done at the leaves, so the latter dominates the total computational complexity.
2. 'Marginal' bookkeeping - dividing and combining is asymptotically as expensive as the work done at the leaves, so neither dominates in the total computational complexity.
3. 'Expensive' bookkeeping - dividing and combining is *polynomially more expensive* than the work done at the leaves, so the former dominates the total computational complexity

n^ε $\frac{1}{n^\varepsilon}$

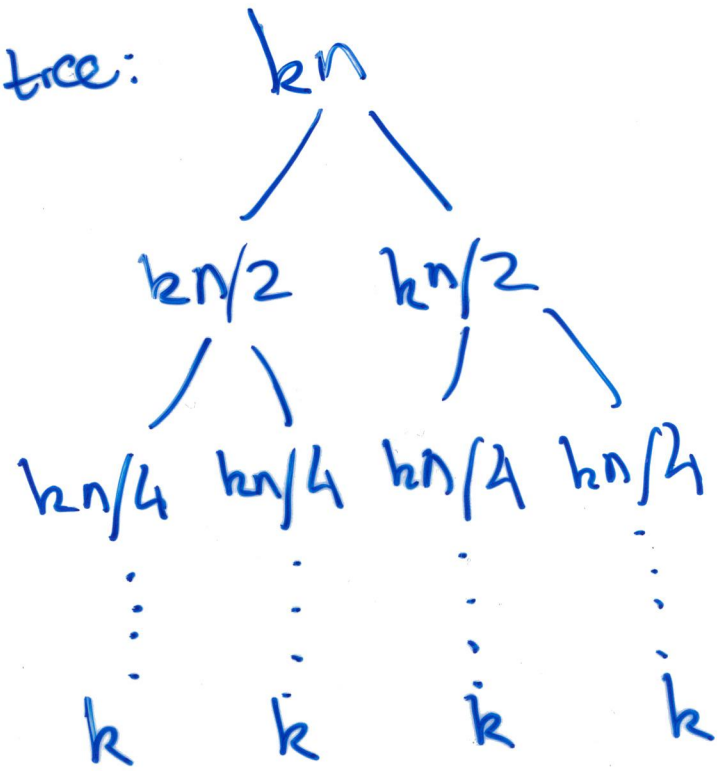
Caveats of the Master Theorem

N.B. *polynomially cheaper* (for example) means just that, n^ε cheaper; logarithmically cheaper is not sufficient.

N.B. condition 3 requires that bookkeeping associated breaking down a problem and combining solutions is cheaper than the bookkeeping associated with the original problem; in other words the costs of bookkeeping do not increase as we go down the recurrence tree.

Applying the Master Theorem - mergesort

Recurrence tree:



$a = 2$

$b = 2$

$f(n) = kn$
$\in \Theta(n)$

$n^{\log_b a}$
$= n^{\log_2 2} = n \in \Theta(n)$

$\therefore$ case 2 applies ("marginal")

$\Rightarrow T(n) \in \Theta(n^{\log_2 2} \log n)$

i.e. $T(n) \in \Theta(n \log n)$

Advanced Programming Topics

Proof by induction, and structural induction
Lecture 5
James Marshall

Testing vs Proof

The usual software engineering approach to checking program correctness is *testing.* Test cases are selected, usually according to some strategy such as

- *black-box testing*
- *white-box testing*

Imagine designing black-box test cases for a function

```
fastIntExp :: Integer -> Integer -> Integer
```

that is supposed to raise its first argument to the power of its second, and return the result.

Test case: a=2, b=2 a=2, b=8 a=2, b=16

Result: 4 ✓ 256 ✓ 65,536 ✓

Now look at the code and design some white-box test cases:

```
fastIntExp :: Integer -> Integer -> Integer
fastIntExp a b
  | b == 1      = a
  | even b      = (fastIntExp a (half b))^2
  | otherwise = (fastIntExp a (b-1)) * a
        where half x = truncate ((fromIntegral x)/2)
```

Test case: a=2, b=1 a=2, b=0 a=2, b=-1 a=2, b=3

Result: 2 ✓ undefined (Should be 1) undefined (Should be $\frac{1}{2}$) 6 ✗

Exercise: fix the `fastIntExp` *function*

Although looking at the code can give us ideas for test cases to pick up particular errors, we still rely on our intuition and experience to generate all the right tests. We may miss some though…

Proof by Induction

To prove that a desired property holds for all *defined* values computed by a function we use *proof by induction*. The major steps are as follows

1. Identify the proof goal
2. Identify the appropriate base case and prove the goal for that case
3. Identify the inductive hypothesis for the inductive step
4. Show that the base case and inductive hypothesis together prove the goal for all defined values of the function

Example - Slow Integer Exponentiation

definitions

Proof goal Prove for all $b \geq 0$ that `slowIntExp a b == a^b`

```
slowIntExp :: Integer -> Integer -> Integer
slowIntExp a 0 = 1
slowIntExp a b = a * (slowIntExp a (b-1))
```

```
--(exp0)
--(exp+)
```

1. Proof goal is as described above

2. Choose base case `slowIntExp a 0 == a^0`

Prove base case:

by arithmetic

by exp0

slowIntExp a 0 == 1

1 == 1 QED

3. Identify inductive hypothesis and inductive step

slowIntExp a (b-1) == a ^ (b-1) ⟹ slowIntExp a b == a^b

4. Show that (base case ∧ inductive hypothesis) ⟹ proof goal

slowIntExp a b == a ^ b

by expt

(by arithmetic)

a * (slowIntExp a (b-1)) == a * (a ^ (b-1))

(simplify)

inductive hypothesis

→ slowIntExp a (b-1) == a ^ (b-1)

Structural Induction

We can also use proof by induction to prove properties of data structures, such as lists.

Example - from Thompson (chapter 8)

Prove, given the following functions

```
sum :: [Int] -> Int
sum []     = 0                          --(sum.1)
sum (x:xs) = x + sum xs                 --(sum.2)

doubleAll []     = []                   --(doubleAll.1)
doubleAll (z:zs) = 2*z : doubleAll zs   --(doubleAll.2)
```

that for all lists `xs`

```
sum (doubleAll xs) == 2 * sum xs        --(sum+dblAll)
```

1. Proof goal is as just described

2. Prove base case:

sum (doubleAll []) == 2 * sum []

(by doubleAll.1) ↓

sum []

(by sum.1) ↓

0

(by sum.1) ↓

2 * 0

(by arithmetic) ↓

0

3. Identify inductive hypothesis and inductive step

sum (doubleAll zs) == 2 * sum zs

⟹ sum (doubleAll (z:zs))

== 2 + sum (z:zs)

4. Show that (base case ∧ inductive hypothesis) ⟹ proof goal

sum (doubleAll (z:zs)) == 2 * sum (z:zs)

(by doubleAll.2) ↓ (by sum.2) ↓

sum (2*z : doubleAll zs) == 2 * (z + sum zs)

(by sum.2) ↓ (by arithmetic) ↓

2*z + sum (doubleAll zs) == 2*z + 2*(sum zs)

Advanced Programming Topics

Proof in the wild: strong induction, imperative proof, automated proof
Lecture 6
James Marshall

Strong Induction

Sometimes we need to consider more than just the next-smallest case in constructing a proof by induction.

e.g. the Fibonacci numbers (from lecture 1)

```
f 1 = 1                          --(fib1)
f 2 = 1                          --(fib2)
f n = f (n-1) + f (n-2)          --(fibn)
```

N.B. from lecture 1 we know that `f 100` takes longer than the universe has been in existence to compute. However, we should be able to prove that the right answer would be given! For example we should be able to prove `fibn`, since we know that the *n*-th Fibonacci number is given by the formula

$$\frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}} \quad \text{where} \quad \varphi = \frac{1+\sqrt{5}}{2}$$

"golden ratio"

Q: …*but*, how can we formulate an inductive hypothesis involving only the previous number in the sequence, when we need to know the previous *two*?

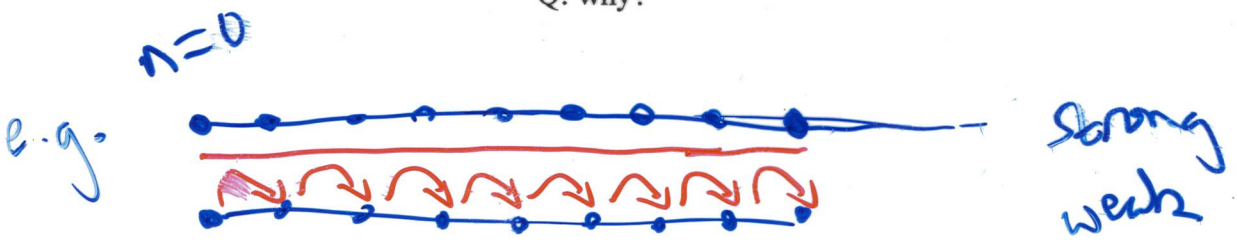
A: by having our inductive hypothesis be that the desired property holds for all $k < n$

Exercise (hard): use the mathematical definition above to prove `f n` *is correct for all* $n \geq 0$

N.B. it may seem that strong induction is more powerful than weak induction, but actually

principle of weak induction $\iff$ principle of strong induction

Q: why?



Strong Induction Example - Integer Exponentiation

```
fastIntExp :: Integer -> Integer -> Integer
fastIntExp a b
    | b == 0     = 1                              --(exp0)
    | even b     = (fastIntExp a (half b))^2      --(expeven)
    | otherwise = (fastIntExp a (b-1)) * a        --(expodd)
    where half x = truncate ((fromIntegral x)/2)
```

Prove that for all defined value `fastIntExp a n == a^n`

1. Prove the base case:

2. Prove the induction step under the inductive hypothesis `fastIntExp a k == a^k` for `k<n`

Case (1), n even

Case (2), n odd

Structural induction is implicitly strong… lists all data structures are list-like

Strong Structural Induction Example - Binary Tree Size (from Thompson chapter 14)

Prove that the number of nodes in a binary tree is strictly less than its 2 to the power of its depth.

```
data NTree = NilT |
             Node Int NTree NTree

depth NilT            = 0                                  --(depth.1)
depth (Node n t1 t2)  = 1 + max (depth t1) (depth t2)      --(depth.2)

size NilT             = 0                                   --(size.1)
size (Node x t1 t2)   = 1 + size t1 + size t2               --(size.2)
```

1. State the proof goal as proving for any `NTree tr` that `size tr < 2^(depth tr)`

2. Prove the base case

size NilT < 2^(depth NilT)
↓ (by size.1)
0 < 2^(0)
↓ (by arithmetic)
0 < 1

3. Prove the induction step, that the property holds for any tree, given the induction hypothesis that the property holds for that tree's left and right subtrees

Inductive step: assuming size tr1 < 2^(depth tr1) AND size tr2 < 2^(depth tr2)

THEN size (Node a tr1 tr2) < 2^(depth (Node a tr1 tr2))
↓ (by size.2)
1 + size tr1 + size tr2 < 2^(1 + max (depth tr1) (depth tr2))
↓ (by depth.2)
↓ worst case: size tr1 == size tr2
↓ (assume tr1 is deepest)
1 + 2 * (size tr1) < 2^(1 + depth tr1)
↓ arithmetic
↓ arithmetic
1/2 + (size tr1) < 2^(depth tr1)

equiv. ind. hyp.

Proving Correctness in Imperative Programs

Most imperative languages do not use recursion, but instead use loops. But loops are very similar to recursion, so we can adapt proof by induction to prove the correctness of loops, by using *loop invariants.* These feature in imperative proof systems such as *Floyd-Hoare logic*, which reason about pre and post-conditions for the execution of functions. Note that fully automated proof is impossible however; if we could specify termination as a post-condition of a function and prove it automatically, we would have solved the *halting problem* (see COM2003 notes).

Definition: Loop Invariant

A *loop invariant* is a property of a loop that satisfies the following conditions (*e.g.* CLRS chapter 2):

1. *Initialisation*: The loop invariant is true before the first iteration of the loop.

2. *Maintenance:* If the invariant is true at the start of a loop iteration then it is also true at the end of the loop iteration.

3. *Termination:* When the loop terminates the invariant helps prove the correctness of the algorithm implemented by the loop.

Loop Invariant Example - Insertion Sort (from chapter 2 of CLRS)

```
INSERTION-SORT(A)
1    for j ← 2 to length[A]
2        do key ← A[j]
3           Insert A[j] into the sorted sequence A[1 .. j − 1]
4           i ← j − 1
5           while i > 0 and A[i] > key
6               do A[i + 1] ← A[i]
7                  i ← i − 1
8           A[i + 1] ← key
```

(Outer) Loop invariant:
At the start of each iteration the subarray $A[1 .. j-1]$ *is a sorted version of the original subarray* $A[1 .. j-1]$

Initialisation:
At initialisaation $A[1 .. j-1] = A[1 .. 1]$ *contains 1 element, so the loop invariant is trivially true*

Maintenance:
Informally, each loop iteration moves the new element to is correct position in the subarray. Formally, could use a loop invariant on the inner loop.

Termination:
When the loop ends $j = n + 1$*, where n is the length of the array to be sorted. By the loop invariant, at termination the array* $A[1 .. j-1] = A[1 .. n]$ *is a sorted version of the original* $A[1 .. n]$

Advanced Programming Topics

Abstract Data Types - Specification and Implementation
Lecture 7
James Marshall

non-numeric

N.B. Abstract Data Types** are not to be confused with **Algebraic Data Types

Why use Abstract Data Types?

Abstract Data Types allow modular software to be designed. They:

- provide a clear *interface* to use the data type (typically called the *signature*, where the functions provided have a defined *syntax*)
- *hide* details of the type's implementation from the user, allowing us to subsequently modify it without breaking code that makes use of it

Axiomatic Specification of ADTs

How should we specify ADTs? We can give an informal definition of how a data structure behaves, but often this will be incomplete, and insufficient for someone without prior knowledge to implement it. The behaviour to be expected by users may also be ambiguous as a result.

With some work, however, we can formally specify the properties of an ADT. If we are careful this can provide a complete and unambiguous description of how the ADT can be used. We can also then use this specification to prove that our implementation of an ADT meets that specification.

Example 1 - Stacks

We wish to specify a *stack* ADT. The *signature* is:

constructors
observers
mutators

- `emptyStack` – constructor
- `isEmpty` – observer
- `top` – observer
- `push` – constructor
- `pop` – mutator

The elements of the signature are divided into three types:

Constructors	Build instances of ADT
Observers	Report on ADT instances w/out changing
Mutators	Change ADT instances

We provide the *syntax* for the elements of the signature as follows:

- `emptyStack :: Stack a`
- `isEmpty :: Stack a -> Bool`
- `top :: Stack a -> a`
- `push :: a-> Stack a -> Stack a`
- `pop :: Stack a -> Stack a`

The *axiomatic specification* of the stack ADT is as follows:

	Axiom	*Meaning*
1.	`isEmpty emptyStack == True --(specIsEmpty.1)`	
2.	`isEmpty (push x s) == False --(specIsEmpty.2)`	
3.	`top (emptyStack) == error "empty stack has no top" --(specTop.1)`	
4.	`top (push x s) == x --(specTop.2)`	
5.	`pop (emptyStack) == error "cannot pop an empty stack" --(specPop.1)`	
6.	`pop (push x s) == s --(specPop.1)`	

N.B. this specification has simpler error handling than the version in Mike Stannett's notes

Example 2 - Sets

Signature:

- `emptySet`
- `isEmpty`
- `isMember`
- `addMember`
- `removeMember`
- `union`
- ...

Syntax:

- `emptySet :: Set a`
- `isEmpty :: Set a -> Bool`
- `isMember :: a -> Set a -> Bool`
- `addMember :: a -> Set a -> Set a`
- `removeMember :: a -> Set a -> Set a`
- `union :: Set a -> Set a -> Set a`

Some Axioms:

- `isEmpty emptySet == True` `--(isEmpty.1)`
- `isEmpty (addMember x s) == False` `--(isEmpty.2)`
- `isMember x emptySet == False` `--(isMember.1)`
- `isMember x (addMember x s) == True` `--(isMember.2)`
- `isMember x (removeMember x s) == False` `--(isMember.3)`
- `(isMember x s) == True || (isMember x r) == True =>`
  `isMember x (union r s) == True` `--(isMember.4)`
- `(isMember x s) == False && (isMember x r) == False =>`
  `isMember x (union r s) == False` `--(isMember.5)`
- `removeMember x (addMember x s) == s` `--(removeMember.1)`
- `union emptySet emptySet == emptySet` `--(union.1)`
- `union emptySet s == s` `--(union.2)`
- `union s emptySet == s` `--(union.3)`

Q: do you think this axiomatic specification is complete for the signature and syntax given?

Implementing ADTs in Haskell (Thompson chapters 15 and 16 for more details)

ADTs are naturally implemented in Haskell using `modules`. In a separate file we define a module, including the functions that are visible by users `importing` that module (*i.e.* the *signature* of the ADT). For the stack example,

```
module Stack (emptyStack, isEmpty, push, pop, top) where

data MyStack a = Top a (MyStack a) | Empty

emptyStack :: MyStack a
isEmpty :: MyStack a -> Bool
push :: a -> MyStack a -> MyStack a
pop :: MyStack a -> MyStack a
top :: MyStack a -> a
```

N.B. if a module importing others wants to hide parts of them from itself and its users, it can; *e.g.*

```
import Stack hiding (pop,isEmpty)
```

is possible, even if it's not that useful. This can be useful if function names clash, for example.

Advanced Programming Topics

Abstract Data Types - Specifications: Completeness, Implementation, Proof
Lecture 8

James Marshall

In lecture 7 it was claimed that "[…] *we can formally specify the properties of an ADT. If we are careful this can provide a complete and unambiguous description of how the ADT can be used.* ***We can also then use this specification to prove that our implementation of an ADT meets that specification.***"

In this lecture we'll have a go at doing this for some simple examples… in principle it's quite simple, we just use the same kind of *equational reasoning* we already used in inductive proofs (*i.e.* rewriting Haskell equations using Haskell function definitions)

But, before we do that, let's look at whether our stack ADT specification was a *complete specification* or not. Here it is again:

```
isEmpty emptyStack == True                                  --(specIsEmpty.1)
isEmpty (push x s) == False                                 --(specIsEmpty.2)
top (emptyStack) == error "empty stack has no top"          --(specTop.1)
top (push x s) == x                                         --(specTop.2)
pop (emptyStack) == error "cannot pop an empty stack        --(specPop.1)
pop (push x s) == s                                         --(specPop.2)
```

Completeness

If the specification is complete, then we should be able to prove other statements about the behaviour of the ADT using the axioms. We can compare these statements against our intuitive understanding of what the ADT should do (if we have one)

Example (from Mike Stannett's notes)

Intuitively, we know that if we have a non-empty stack, if we pop something then push it straight back on, we're left with the original stack. This isn't one of the axioms specifying the stack ADT, so can we prove it using those axioms? *I.e.* we try to prove that

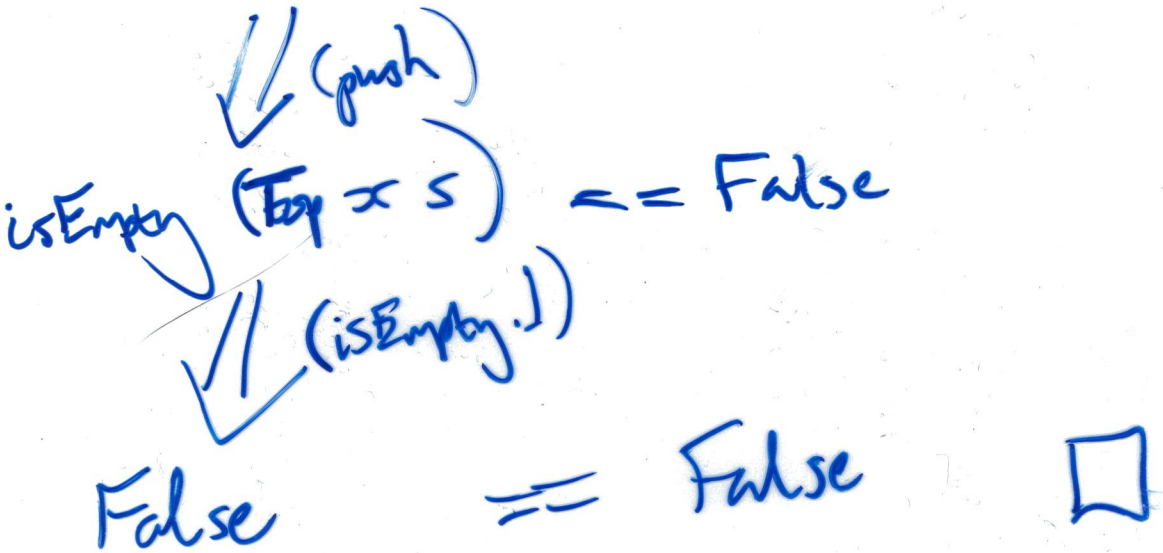
```
isEmpty s == False => push (top s) (pop s) == s
```

Example 1 - Stack Implementation

```
module Stack (emptyStack, isEmpty, push, pop, top) where

data MyStack a = Top a (MyStack a) | Empty

emptyStack :: MyStack a
emptyStack = Empty                          --(empty)

isEmpty :: MyStack a -> Bool
isEmpty (Top x s) = False                   --(isEmpty.1)
isEmpty Empty = True                        --(isEmpty.2)

push :: a -> MyStack a -> MyStack a
push x s = Top x s                          --(push)

pop :: MyStack a -> MyStack a
pop (Top x s) = s                           --(pop.1)
pop Empty =                                 --(pop.2)
  error "Cannot pop an empty stack"

top :: MyStack a -> a
top (Top x s) = x                           --(top.1)
top Empty =                                 --(top.2)
  error "No top on an empty stack"
```

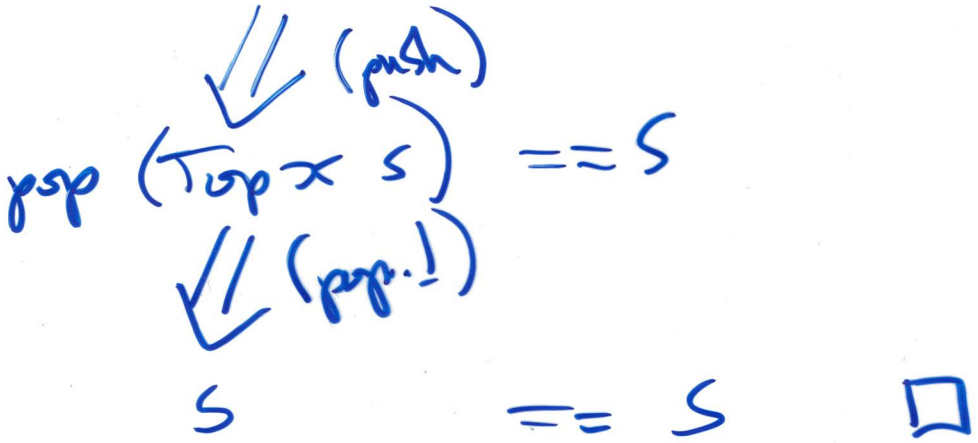
Prove that the implementation satisfies the stack ADT's axiom

```
isEmpty (push x s) == False
```



Prove that the implementation satisfies the stack ADT's axiom

```
pop (push x s) == s
```



Example 2 - Sets

```
• isEmpty emptySet == True                                  --(specIsEmpty.1)
• isEmpty (addMember x s) == False                          --(specIsEmpty.2)
• isMember x emptySet == False                              --(specIsMember.1)
• isMember x (addMember x s) == True                        --(specIsMember.2)
• isMember x (removeMember x s) == False                    --(specIsMember.3)
• (isMember x s) == True || (isMember x s) == True =>
    isMember x (union r s) == True                          --(specIsMember.4)
• (isMember x s) == False && (isMember r s) == False =>
    isMember (union r s) == True                            --(specIsMember.5)
• removeMember x (addMember x s) == s                       --(specRemoveMember.1)
• union emptySet emptySet == emptySet                       --(specUnion.1)
• union emptySet s == s                                     --(specUnion.2)
• union s emptySet == s                                     --(specUnion.3)
```

Exercise: try to prove some of the above axioms are satisfied by the implementation on the COM2001 homepage

But wait, is our specification even complete?

Intuitively, membership of a set shouldn't depend on what order we inserted and removed things form that set. But could we prove (for example) that

```
isMember 1 (insertMember 2 (removeMember 1 emptyStack)) == False
```

using only the axioms above?

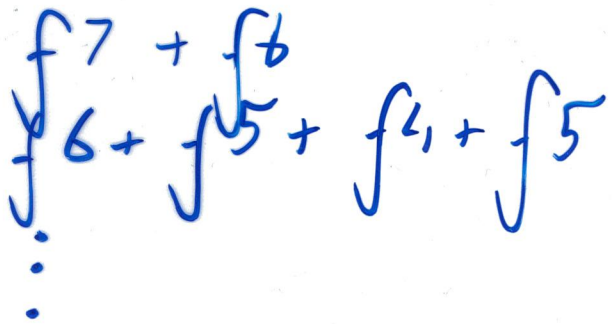
Advanced Programming Topics

Algorithm analysis and design - dynamic programming
Lecture 9
James Marshall

In lecture 1 we saw an efficient solution to computing the Fibonacci numbers, and an inefficient solution

But what made the inefficient solution inefficient?

Trace: `f 8` =



And the efficient solution efficient?

Remember answers (memoize)

N.B. this change turned an exponential-time algorithm into a polynomial-time one

Dynamic Programming (chapter 15 in CLRS)

Dynamic Programming is a technique for efficient solution of problems having:

1. *(Optimal) substructure* —

the (optimal) solution to a problem contains the (optimal) solutions to subproblems ('*optimal*' refers to the frequent usage of dynamic programming in optimisation problems (next lecture))

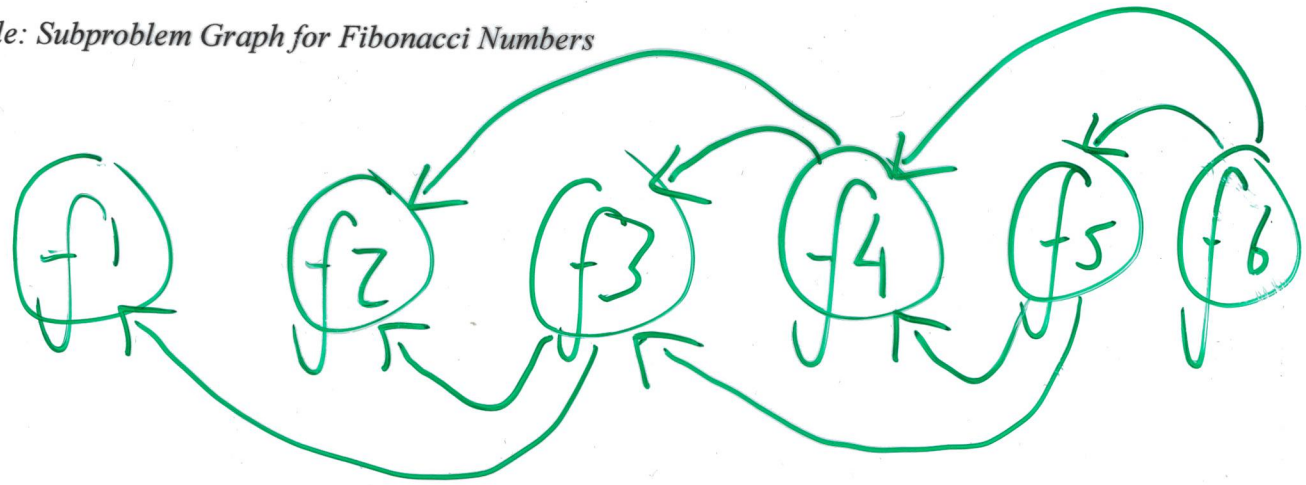
2. *Overlapping independent subproblems* —

subproblems form part of the solution to multiple other problems (*cf.* 'regular' divide-and-conquer problems in which each divide step generates new and non-overlapping subproblems)

Subproblem Graphs

Like *recursion trees*, but *subproblem graphs* show the relationships between all the *distinct* subproblems (vertices), with a directed edge from problem x to problem y showing that a solution to problem x requires a solution to problem y

Example: Subproblem Graph for Fibonacci Numbers



Identifying and Implementing Dynamic Programming Algorithms in Haskell

To identify that a dynamic programming algorithm may solve a problem, and then implement one:

1. Determine whether the description of the solution to a problem shows that the problem exhibits (optimal) substructure (see above) — *e.g.*

"if we work out that the quickest train route from London to Sheffield includes a stop in Derby, we have also worked out the quickest train route from London to Derby"

2. Determine whether the subproblems can be solved *independently* (does solving one subproblem affect how we solve a different subproblem? Often it doesn't, but sometimes it does…) — *e.g.*

"working out the optimal route from London to Birmingham doesn't change how we work out the optimal route from London to Nottingham"

3. Determine whether solutions to subproblems are used in multiple different solutions (if not then simple recursion will suffice) — *e.g.*

"in working out the quickest train route from London to Sheffield we evaluate multiple routes that all require us to know the quickest route from London to Derby"

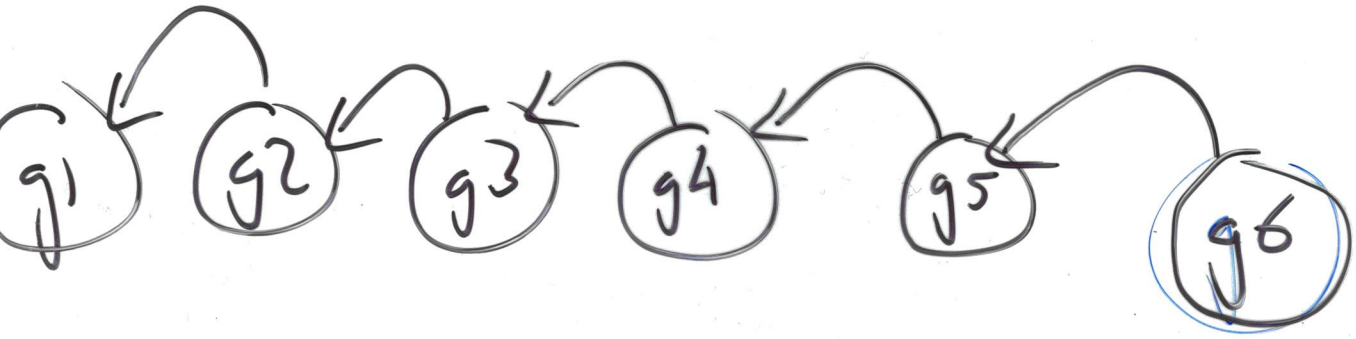
4. Implement using *top-down recursion with memoization*

   A. start by asking for the answer to the original question (e.g. the 100th Fibonacci number), defined recursively in terms of the answers to smaller questions - hence *top-down*

   B. when a value is computed, it is *memoized* (stored) in an efficient data structure

   C. when a value is required, it is searched for in the structure first; if not found then it is computed recursively

Many languages let you define functions as being automatically memoized. Alternatively, use a data structure to store computed values, *e.g.* Data.Map in Haskell. For simple enough recursions (e.g. the Fibonacci example of lecture 1) it may be possible to keep track of a small number of memoized values at each step in the recursion



Advanced Programming Topics

Algorithm analysis and design - dynamic programming and optimisation
Lecture 10
James Marshall

In the last lecture was saw *dynamic programming* used for efficiently computing the results of functions. The technique originates in *decision optimisation*, and the principle of optimal substructure was formalised by Richard Bellman in what is now known as the *Bellman equation.*

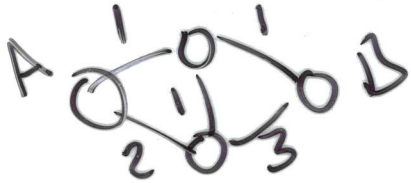
Definition - the Bellman Equation

$$V(x) = \max_a \left(F(x,a) + \beta V(x') \right)$$

V: value function; x: current state; a: possible actions; $F(x,a)$: payoff of a in state x; x': resulting state; β: discount factor in $[0,1]$

Examples

Example 1 - Getting from A to B (Dijkstra's Shortest Path Algorithm)



Naive solution: try all possible paths from A to B

Up to $n!$ paths $\Rightarrow O\left(\sqrt{n}\left(\frac{n}{e}\right)^n\right)$ (Stirling's approximation)

Algorithm idea: if C is on the shortest path from A to B it doesn't matter how you got there (optimal substructure *and* overlapping subproblems (see naive solution; all possible paths from A to B also includes all possible paths from A to C and from C to B))

$$V(x) = \min_a \left(F(x,a) + V(x') \right)$$

$V(x)$: cost of shortest path from x; a: possible edges; $F(x,a)$: cost of edge a from x; x': node reached via a

Algorithm overview: for each node store the best total cost to get to that node so far, and the preceding node on that path; successively update these. At each update step choose the lowest total distance unvisited node (see 'greedy algorithms' below)

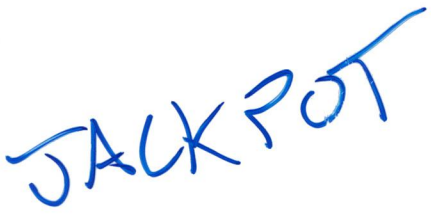
Up to $n(n-1) \Rightarrow O(n^2)$

N.B. $\beta = 1$

Example 2 - Gambling (One-armed bandits and Gittins indices)

Imagine you are in a casino lobby somewhere like Las Vegas:

- In the lobby you are choosing between two one-armed bandits to play, which you believe have different probabilities of paying out on each pull of the arm (trial)
- You used to work for the manufacturer so you know the distribution of success probabilities of these kinds of machines
- You have a pile of dimes to play with - for each dime you have to decide which bandit to put it in
- You only get information about the bandits' payout probabilities by playing them, and given how valuable future payouts to you are (β in the Bellman equation) you want to maximise your long-term discounted reward by eventually settling on only playing the highest probability bandit
- Bayes' rule and dynamic programming you can give you the provably optimal strategy (calculating Gittins indices)… in other words, the most (expected) cash possible!



Optimisation: Dynamic Programming vs Greedy Algorithms

In optimisation problems, at each step in constructing a solution a *greedy algorithm* makes the *locally best choice*; this can be referred to as a *greedy choice* or a *myopic choice*

Greedy algorithms can have advantages and disadvantages:

- Greedy algorithms are usually (much) faster than dynamic programming algorithms…
- …but may not guarantee the optimal solution will be constructed, *although they can be shown to for many important problems, such as computing shortest paths (e.g. Dijkstra's algorithm)!*

N.B. just because a greedy algorithm exists to optimally solve a problem, that does not mean *any* greedy algorithm for that problem will lead to an optimal solution. *E.g.* "*take the next train departing London St Pancras*" would be a greedy route-planning algorithm, but not one guaranteed to get you where you want to go quickly!